

# ECE438

## Design of Computers

Instructor: Andrew David Targhetta

March 11, 2020

# Introduction

- ▶ Most material will come from chapter 4
- ▶ The primary focus will be on the *microarchitecture* aspect of computer design
- ▶ We will start by looking at a *single-cycle* MIPS implementation
- ▶ Then move on to a *pipelined* MIPS implementation

## The *microarchitecture*...

- ▶ The control and datapath portion of the processor
- ▶ The implementation of the ISA
- ▶ Determines CPI and cycle time
- ▶ Related terms: *pipelining*, *superscalar*, *out-of-order*, etc.

# A Basic MIPS Implementation

- ▶ Start with a subset of the MIPS ISA to include:
  - ▶ Memory reference instructions: *lw*, *sw*
  - ▶ Arithmetic/logical instructions: *add*, *sub*, *and*, *or*, *slt*
  - ▶ Control flow instructions: *beq* and *j*
- ▶ Build a *single-cycle*, 32-bit implementation to help understand the **datapath**

# Instruction Execution

- ▶ A lot of similarity exists between instructions, which was part of the RISC concept
- ▶ Nearly every instruction must:
  - ▶ Send the **Program Counter (PC)** to **instruction memory** in order to *fetch* an instruction
  - ▶ *Read one or two registers* from the **register file**, using fields in the instruction to select which registers to read
- ▶ Can you think of an instruction that does not have to perform one of the above steps??

# Instruction Execution

- ▶ The remaining steps depend on the instruction type. For example a *lw*:
  - ▶ The **Arithmetic-Logic Unit (ALU)** is used to calculate the address
  - ▶ The **data memory** is *read* using the calculated address
  - ▶ The *word* (32-bit value) read from memory is written to the **register file**
- ▶ Arithmetic/logical instructions do not use the data memory, but do use the ALU and write to the register file

## A brief reminder of logic design

- ▶ Our datapath is a mix of combinational and sequential elements
- ▶ For *combination* logic, the output depends solely on the inputs
- ▶ For *sequential* logic, the output depends on the input and the *state*

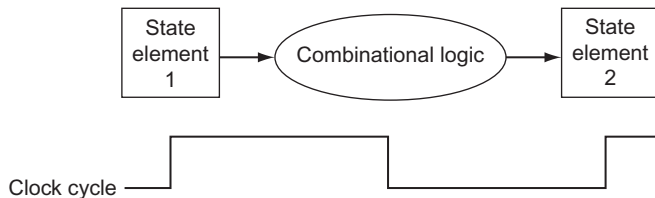
# Combinational and Sequential logic

- ▶ The ALU is a fine example of purely combinational logic:
  - ▶ The output (a 32-bit word) depends solely on the inputs (two 32-bit words and a control bus)
  - ▶ Constructed from various logic gates
- ▶ The register file is a good example of sequential logic with state elements:
  - ▶ The outputs (2 32-bit words) depend on the inputs (two 5-bit register selects) and the state of the register file (the 31 32-bit read/write registers)
  - ▶ Constructed from various logic gates and D flip-flops



## Logic review continued...

- ▶ Our datapath uses an *edge-triggered clocking* methodology
- ▶ State elements (i.e. D flip-flops and memory) are updated on the *rising edge* of the clock



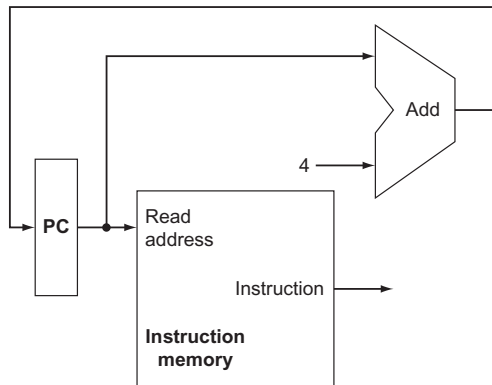
# Want more review??

- ▶ Appendix B of your textbook has a great review of the digital logic elements we will use in this class
- ▶ We can go over more of this in office hours

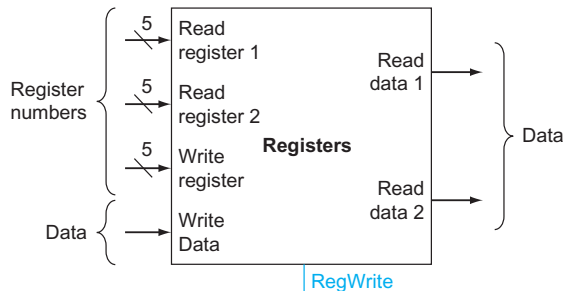
# Okay back to the datapath...

- ▶ Let's start with *instruction fetch*
- ▶ Ignoring jumps and branches for now!
- ▶ Remember we must:
  - ▶ Feed the PC into the instruction memory
  - ▶ Read the instruction memory
  - ▶ Update the PC by adding 4

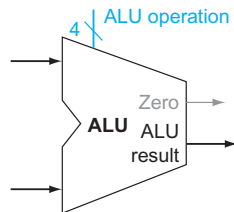
# Instruction Fetch



# Now we need a register file and ALU

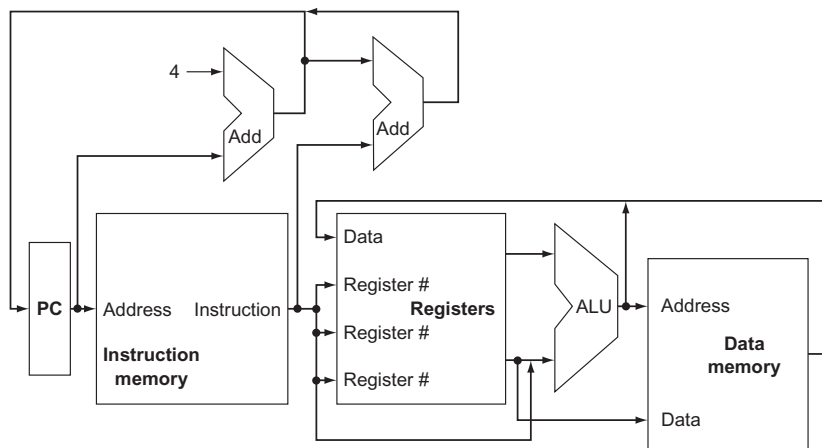


a. Registers



b. ALU

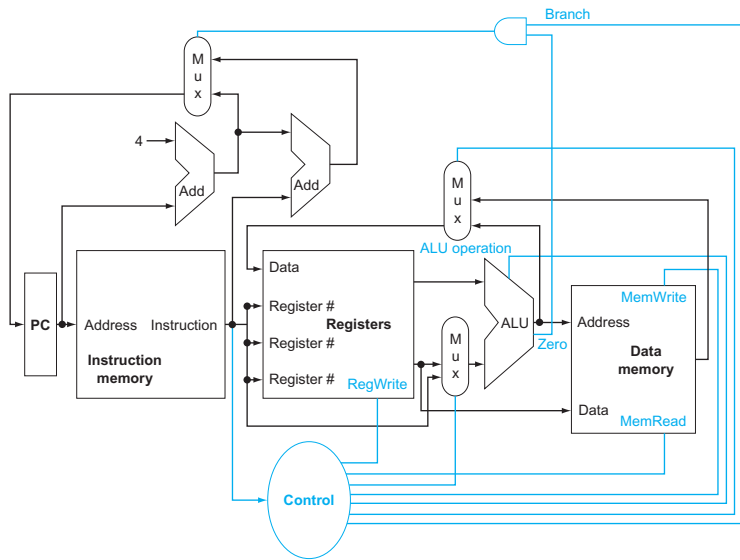
# Combine with a data memory...



# What is missing??

- ▶ The previous figure is missing control and multiplexers (a.k.a. muxes)
- ▶ Muxes allow selection of various data paths
- ▶ Control orchestrates the flow of data through the various data paths (i.e. through the datapath!)

# Combine with control and muxes





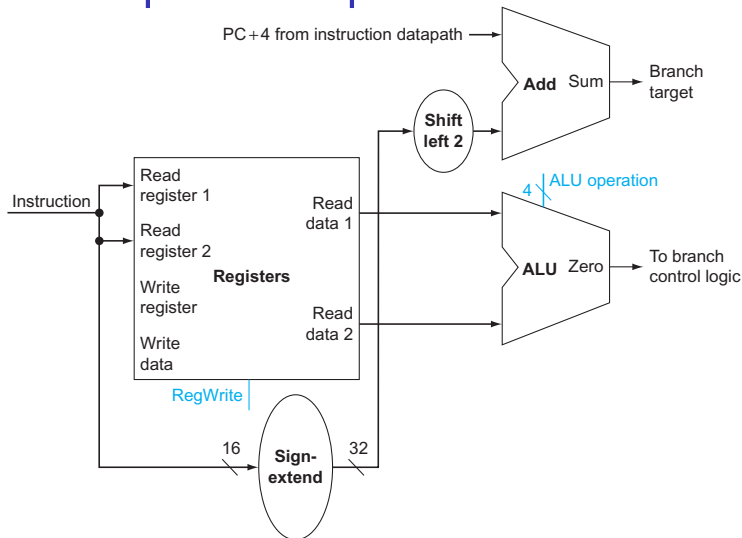
# The previous datapath is still missing some things...

- ▶ Can we identify them?

# The previous datapath is still missing some things...

- ▶ Can we identify them?
- ▶ Sign extension unit
- ▶ Shift left by 2 unit

# Extra datapath components for Branches



# A reminder on instruction format

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

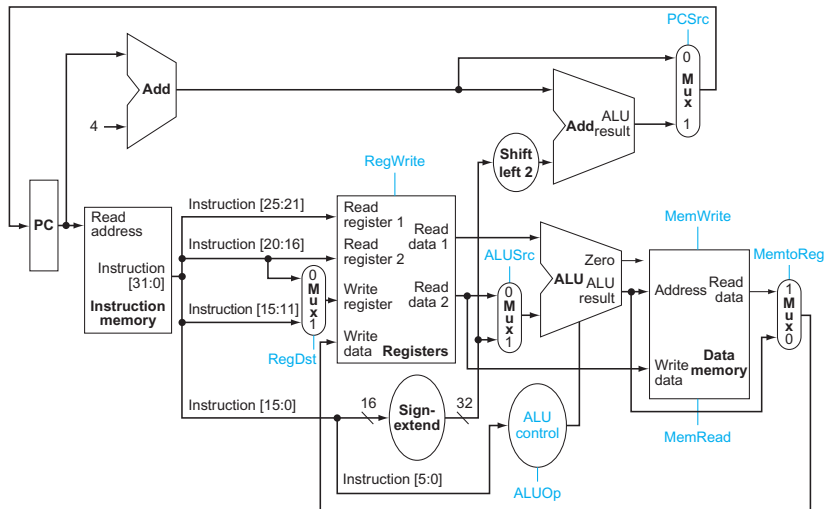
Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

## Key observations

- ▶ The *opcode* is always bits 31:26
- ▶ The 2 regs to be read are *always* specified by *rs* and *rt*
- ▶ The base register for a *lw*, *sw* is *always* specified by *rs*
- ▶ The 16-bit offsets for branches, loads, and stores is *always* in 15:0
- ▶ The destination register is either *rd* for R-type instructions or *rt* for I-type

# Datapath so far



## Notable features...

- ▶ Control signals are identified in light blue
- ▶ There is a mux for specifying the **write register**
- ▶ There is now this ALU control unit

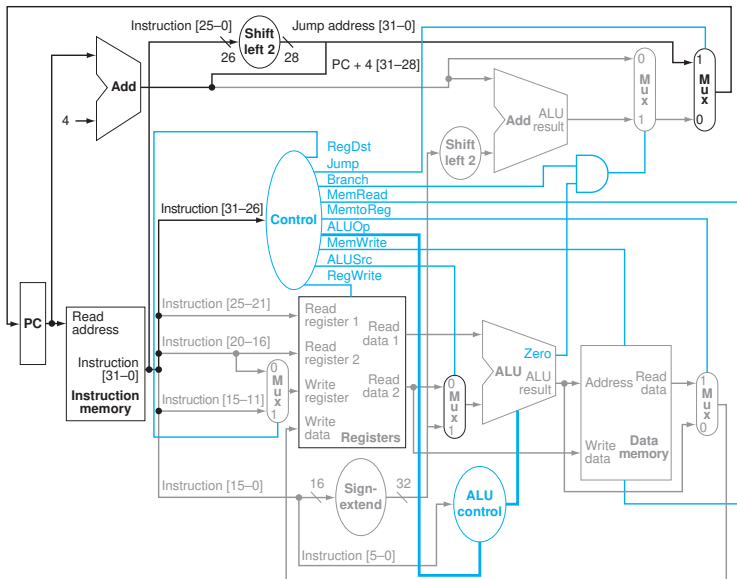
# The datapath *still* does not support some of the instructions we've discussed so far

- ▶ Jumps! such as *j*, *jr*, *jal*, *jalr*
  - ▶ Need another mux input to the PC for jump address
  - ▶ Need a mux to select between the different jump targets
  - ▶ Need to be able to set **write register** to 31
- ▶ Shifts such as *sll*, *srl*, *sra*
  - ▶ Need a mux on the A input of ALU to mux in *shamt*



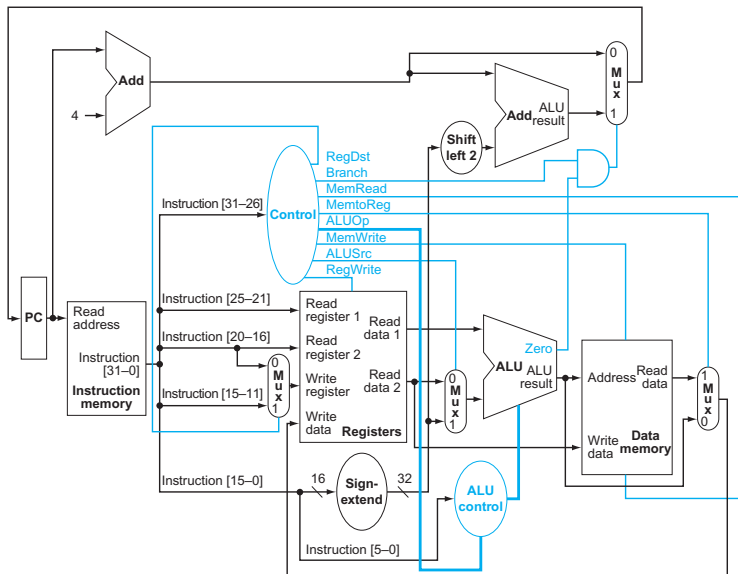
# Let's fix the datapath...

- ▶ Redraw figure 4.15 from book on to white board
- ▶ Add in new datapaths



# Let's look at the control signals...

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of $PC + 4$ .	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



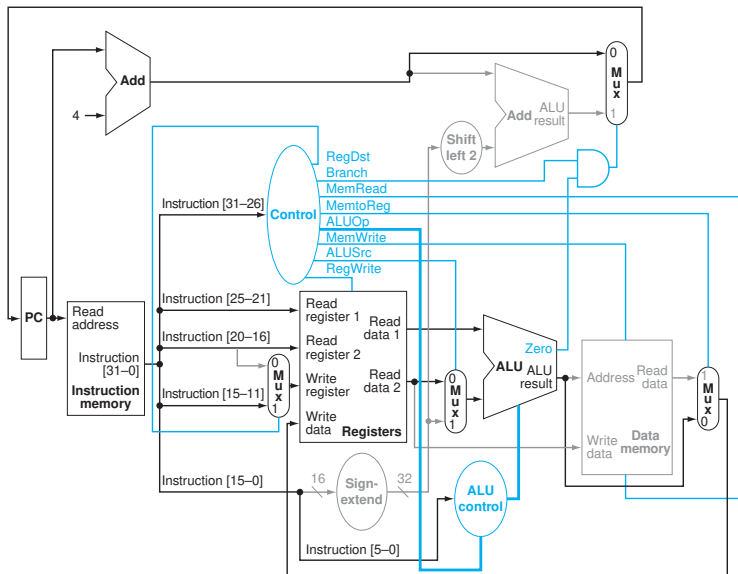
# How the ALU control bits are set

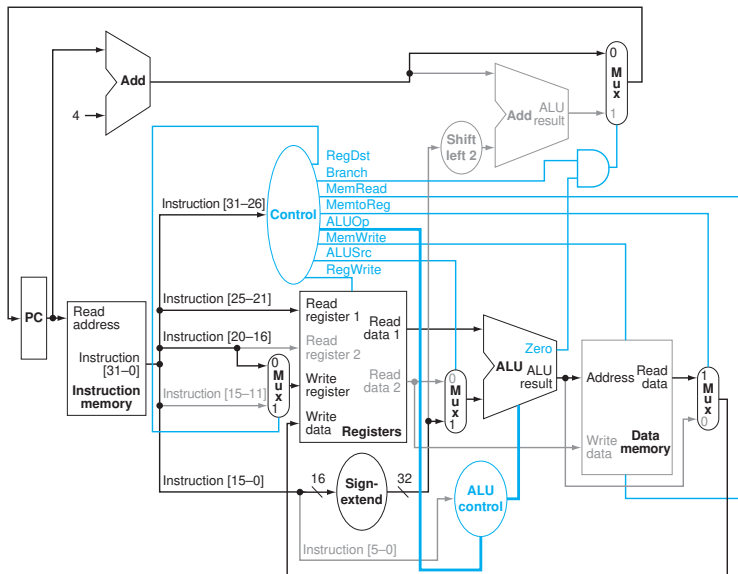
Instruction opcode	ALUOp	Instruction operation	Func field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

# Setting the control signals based on the *op* field

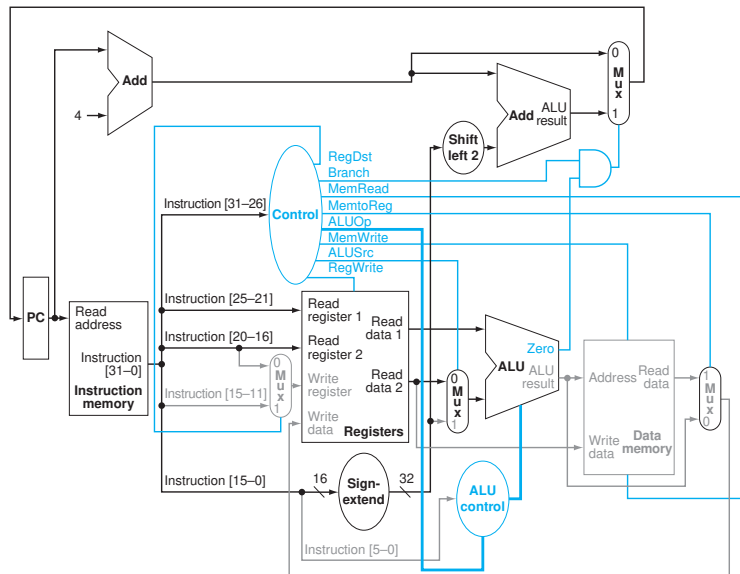
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- ▶ Xs represent don't cares, which means that part of datapath is unused
- ▶ The above table groups all R-type instructions together, which does not quite work for a real MIPS implementation



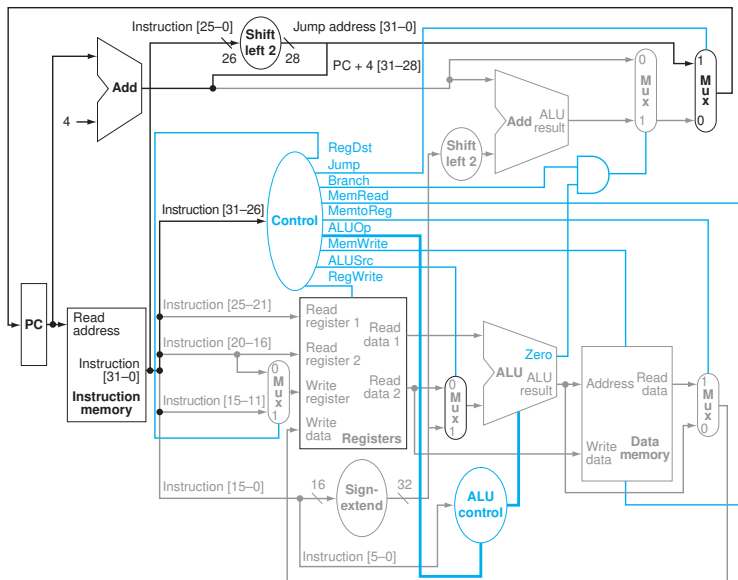






# Simple truth table for control

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



# Why is a *single-cycle* implementation impractical??

- ▶ The cycle time of such a processor would be quite large
- ▶ The longest instruction,  $I_w$ , would dictate the **critical path**
- ▶ It's wastes energy, leaving a large portion of the datapath inactive during any give cycle

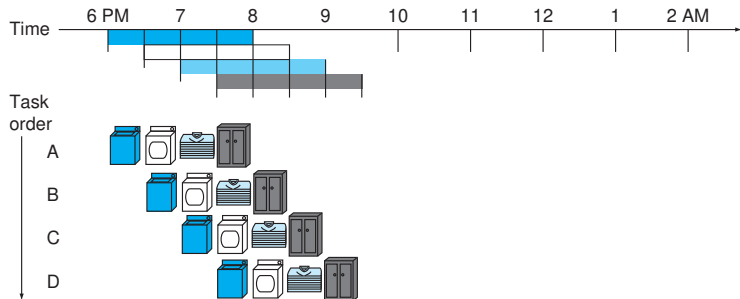
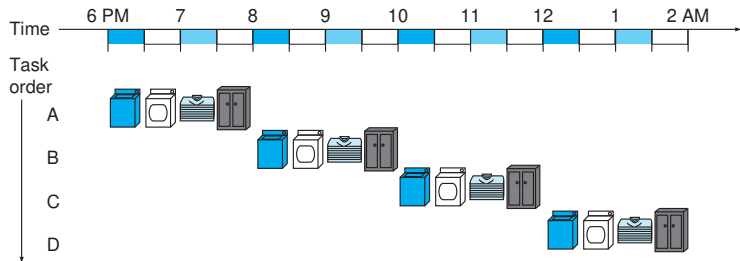
# What can we do instead??

## Pipeline the processor!

- ▶ Break the datapath up into sections with registers between each section
- ▶ Allow an instruction to start every cycle
- ▶ Complicates the control logic and introduces hazards!

# Think about how you do laundry!

- ▶ Place a load in washer
- ▶ When washer finishes, put in dryer
- ▶ When dryer finishes, fold
- ▶ Put clothes away when done



## The laundry example...

- ▶ If each step takes 30 minutes, it would take 8 hours to do 4 loads of laundry sequentially!
- ▶ If we pipeline the operation, it only takes 3.5 hours to do 4 loads
- ▶  $\text{speedup} = 8/3.5 = 2.29$
- ▶ Speedup improves as we do more loads of laundry.
- ▶ Speedup goes to 4x as we do infinite loads



## We can break the datapath up similarly

- ▶ Fetch instruction and add 4 to PC
- ▶ Decode instruction and read register file
- ▶ Execute ALU operation
- ▶ Access memory if needed
- ▶ Write result into register file

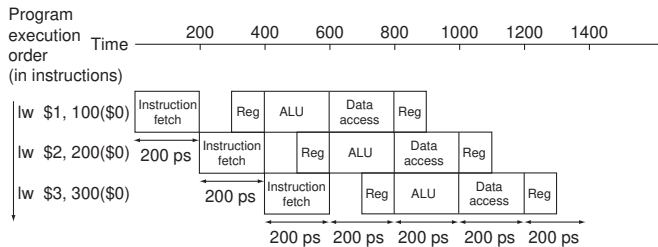
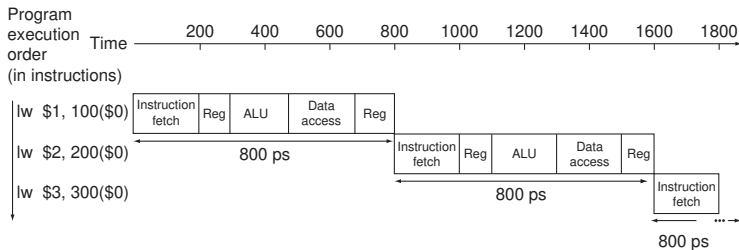
# The speedup of pipelining

- ▶ In the *ideal case*, the speedup of pipelining is equal to the number of pipeline stages!
- ▶ So what is the ideal case?
  - ▶ The stages are perfectly balanced, so the clock rate is  $n$  times faster if  $n$  is the number of stages
  - ▶ The instruction stream is infinite so we can ignore the filling and draining of the pipeline
  - ▶ There are no pipeline *hazards*, i.e. dependencies amongst instructions
  - ▶ The last 2 imply that a  $CPI=1$  is maintained

# Components within the critical path

- ▶ The critical path is the longest path which determines the cycle time
- ▶ For single-cycle implementation, the critical path is determined by  $lw$
- ▶ For pipelining, it's the longest pipeline stage

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word ( $lw$ )	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word ( $sw$ )	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, $sllt$ )	200 ps	100 ps	200 ps		100 ps	600 ps
Branch ( $beq$ )	200 ps	100 ps	200 ps			500 ps



## In the previous figure...

- ▶ Executing 3 instructions sequentially takes 2400ps, while pipelining them takes 1400ps
- ▶ This is a speedup of  $2400ps/1400ps = 1.7$
- ▶ As instruction count moves out to infinity, speedup approaches  $800ps/200ps = 4$ , assuming no *hazards*!

## Some things to keep in mind...

- ▶ The previous calculation only accounts for the increase in clock rate and ignores the inevitable increase in CPI
- ▶ Pipelining increases instruction *throughput* as opposed to decreasing instruction *latency*
- ▶ In fact, it typically *increases* instruction latency!
- ▶ In our example, a *lw* takes 800ps sequentially, and 1000ps in the case of pipelining

## How does RISC make pipelining easier??

- ▶ Fixed instruction length means we can *fetch* the instruction in 1<sup>st</sup> cycle and decode the instruction in 2<sup>nd</sup>. With variable length instructions, the fetching and decoding is intertwined. Why?
- ▶ Fewer instruction formats with symmetry between them means that register file can be read while decoding the instruction. In a CISC ISA, we would have to partially decode instruction before reading register file

## How does RISC make pipelining easier??

- ▶ RISC uses load-store concept so ALU stage can be used for address calculation. If mem-reg style ALU instructions were allowed, the ALU stage would have to follow the memory stage and additional address logic would be required.
- ▶ Finally, memory alignment means we can access a data word in 1 cycle as opposed to multiple cycles



# What are pipeline hazards??

- ▶ A *hazard* is a situation that has the potential to interrupt the flow of instructions through the pipeline
- ▶ There are three types of hazards:
  - ▶ A *structural hazard* exists when instructions in the pipeline are competing for the same resource
  - ▶ A *data hazard* exists when there are data dependencies between nearby instructions
  - ▶ A *control hazard* exists due to branches and jumps, i.e. control flow instructions

## How do we resolve structural hazards??

- ▶ An example of a structural hazard in our simple pipeline involves the access of memory in both the *fetch* and *memory* stages
- ▶ We resolve this structural hazard by duplicating the memories
- ▶ More complicated pipelines that support floating point must deal with many more structural hazards by either stalling some instructions or duplicating resources

## What about data hazards??

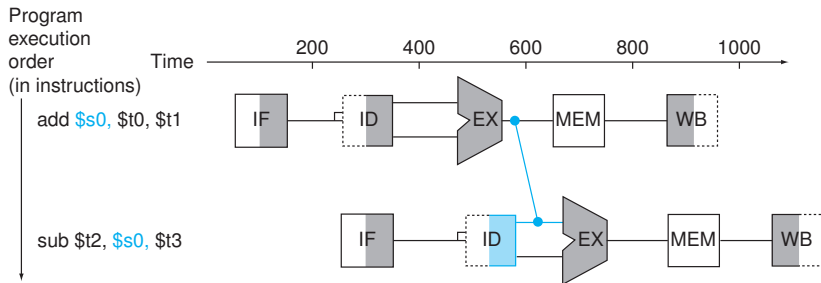
- ▶ An example data hazard:

**add** \$s0 , \$t0 , \$t1

**sub** \$t2 , \$s0 , \$t3

- ▶ The **add** is *producing* the value \$s0, while the **sub** is *consuming* \$s0
- ▶ The issue here is that the **add** doesn't write \$s0 until the *write-back* stage, while the **sub** reads \$s0 during the *decode* stage

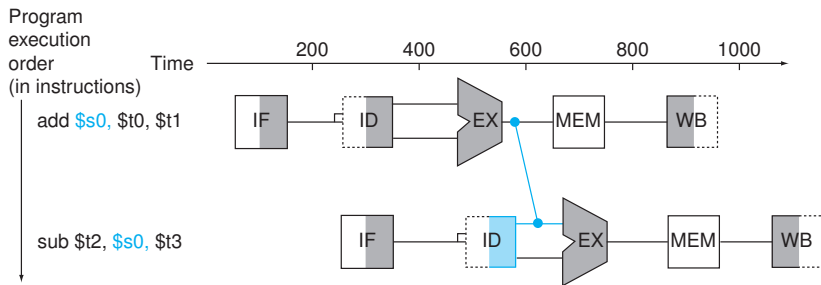
# Visualize the data hazard!



## How to resolve this hazard?

- ▶ We can either stall the execution of the **sub** until the data is written into the register file
- ▶ How many cycles would we have to stall?
- ▶ Or we can *forward* the data from the **add** to the **sub** without waiting for it to be written and then read into the register file

- Observe the fact that data is available when the **add** finishes the *execute* stage and isn't needed until the **sub** begins its *execute* stage



## Data hazards continued...

- ▶ Forwarding essentially bypasses the register file and thus is also referred to as *bypassing*
- ▶ Note that `$s0` will still be written into the register file so other instructions can use it
- ▶ For back-to-back ALU instructions, no intervening stalls are necessary, i.e. forwarding completely resolves the data hazard

## Is this always true??

- ▶ The following is an example of a data hazard that cannot be completely resolved with forwarding:

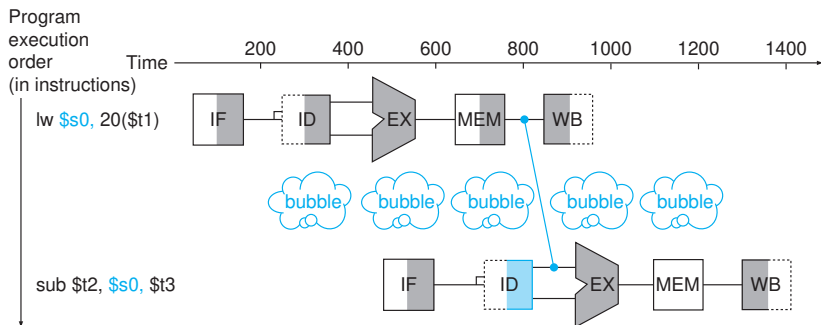
```
lw  $s0 , 20($t1)  
sub $t2 , $s0 , $t3
```

- ▶ The issue here is that **lw** does not produce the data until the end of the *memory* stage, which is one cycle later than when the **sub** needs it



# So then what do we do?

- We stall the **sub** one cycle and then forward data from the memory stage to the execute stage



## Can we avoid these types of stalls?

- We can! For example, consider the code below:

Handwritten red annotations on the assembly code:


- Red circles around `$t1`, `$t2`, `$t3`, `$t4`, and `$t5` in the instructions.
- Red arrows pointing from `$t1` in the first instruction to `$t1` in the third and fourth instructions.
- Red arrows pointing from `$t2` in the second instruction to `$t2` in the third instruction.
- Red arrows pointing from `$t3` in the third instruction to `$t3` in the fourth instruction.
- Red arrows pointing from `$t4` in the fifth instruction to `$t4` in the sixth instruction.
- Red arrows pointing from `$t5` in the sixth instruction to `$t5` in the seventh instruction.
- Red checkmarks next to the `add` and `add` instructions.
- Red text "nop" with an arrow pointing to the space between the second and third instructions.
- Red text "nop" with an arrow pointing to the space between the fifth and sixth instructions.

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

- There are 2 stalls above. Where?

## Can we avoid these types of stalls?

- We can rearrange the code to avoid stalls:



```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

# What about control hazards?

- ▶ Consider this:
  - ▶ The pipeline does not recognize a jump instruction until the *decode* stage
  - ▶ However, it must know where in memory to fetch the instruction following the jump during the *fetch* stage
- ▶ This dilemma is an example of a control hazard!

## But wait! There's more...

- ▶ Jumps are easy compared to branches:
  - ▶ Branches have a condition to *resolve*
  - ▶ In our example, the branch condition is not resolved until the *execute* stage
- ▶ So where does the pipeline fetch from in the meantime?
- ▶ The easy thing to do is continue fetching sequential ( $PC + 4$ ) until the branch is resolved, i.e. “assume not taken”

## Resolving control hazards

- ▶ For both branches and jumps, we can continue fetching sequentially until we decode the jump or resolve the branch
- ▶ However, our pipeline must be able to *flush* instructions that were fetched down the wrong path and redirect fetching after a jump is decoded or a branch is resolved

## Penalties for control hazards

- ▶ For a jump, the pipeline needs to flush the instruction immediately following, so that would be a one cycle penalty
- ▶ For a branch, there could be two situations:
  - ▶ If the branch resolves to *Not Taken*, fetching sequentially was the correct thing to do so no penalty
  - ▶ If the branch resolves to *Taken*, the pipeline needs to flush the instruction in the *fetch* and *decode* stages, so a two cycle penalty

# Does it get worse?

- ▶ For deeper pipelines (i.e. more stages), we may have to flush more instructions!
- ▶ For superscalar architectures that fetch more than one instruction per pipeline, we could flush dozens of instructions for a branch!



## What else can we do?

- ▶ For our simple pipeline, we could just unconditionally execute the instruction immediately following a jump or branch
- ▶ The penalty for a jump would be zero cycles and a Taken branch would be one cycle
- ▶ This technique delays the fetching decision for a jump or a branch by one cycle and is appropriately referred to as a *branch delay slot*

## The branch delay slot

- ▶ Hidden from the programmer
- ▶ The assembler will attempt to find a useful instruction to put into the delay slot
  - ▶ An instruction that can execute regardless of the outcome of the branch
  - ▶ A **nop** if nothing else
- ▶ Only one delay slot was ever used
- ▶ Made less sense for deeper pipelines and even less for superscalar architectures
- ▶ RISC-V eliminated the delay slot

IF	I/O	EX
<del>ST</del>	SW	beg

beg st1, st2 delay slot  
 SW  
 ST

## Branch prediction

- ▶ We can further reduce the penalty of a branch by predicting the outcome
- ▶ If the *prediction* was wrong, we flush the pipeline and begin fetching in the correct location
- ▶ Predictions can be *static*
  - ▶ Static prediction could be based on the direction of the branch, i.e. backwards taken, forward not taken
  - ▶ Branch *likely* instructions allow the compile to provide hints or best guesses

# Branch prediction continued

- ▶ Predictions can be *dynamic*
  - ▶ The pipeline can keep track of the branch *history* and make a guess based on what the current branch and branches around it have previously done
  - ▶ The performance of a modern processor is based on accurate branch prediction
  - ▶ Building accurate branch predictors is a very important and active field within computer architecture

b eq \$t0, \$t1 else

ori

j Exit

else: xor

Exit:

if (\$t0 != \$t1)

ori

else  
xor

## A quick summary on pipelining

- ▶ Pipelining is a form of *instruction-level parallelism* because it overlaps the execution of instructions
- ▶ We strive for a  $CPI = 1$  but hazards get in the way
- ▶ We strive for a clock rate increase of  $n$ , which is the number of pipeline stages, but *internal fragmentation* keeps us from perfectly dividing up the pipeline

# Let's look at *how* to pipeline

- ▶ Break the single-cycle design into the 5 stages
- ▶ Add registers between stages



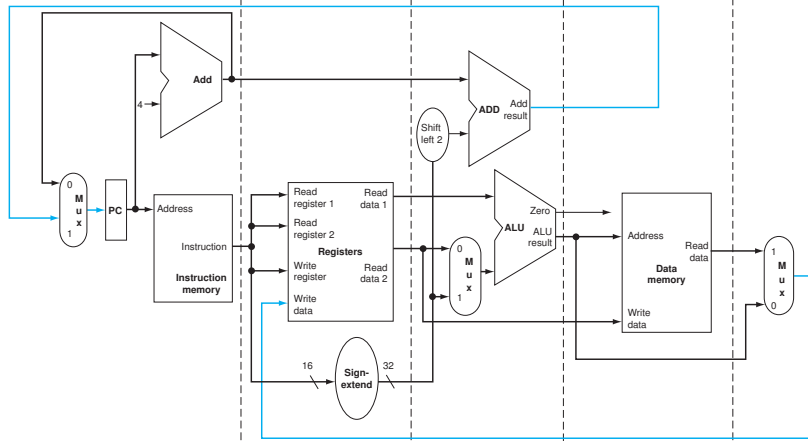
# Break the single-cycle design into stages

IF: Instruction fetch

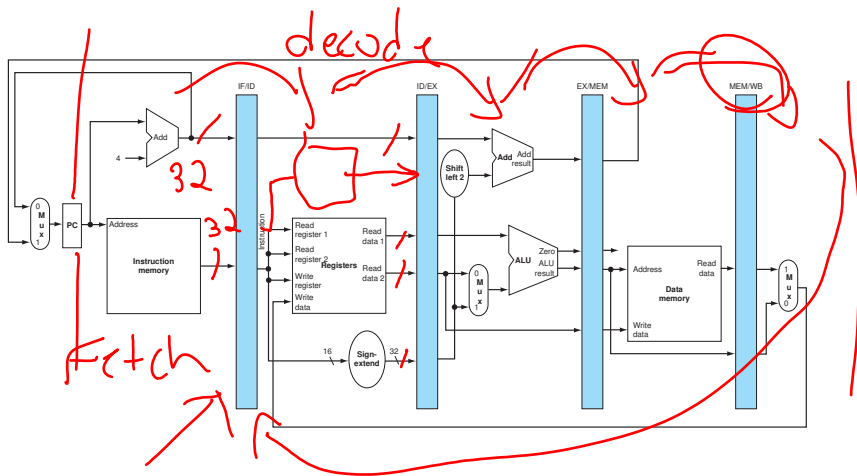
ID: Instruction decode/  
register file readEX: Execute/  
address calculation

MEM: Memory access

WB: Write back



# Add registers between stages



## The previous diagram...

- ▶ Is missing a lot of detail that we will fill in again later
- ▶ Has pipeline registers named based on the stages they separate, so *IF/ID* for example separates the Instruction Fetch and Instruction Decode stages
- ▶ Shows the state of the pipeline in 1 clock cycle

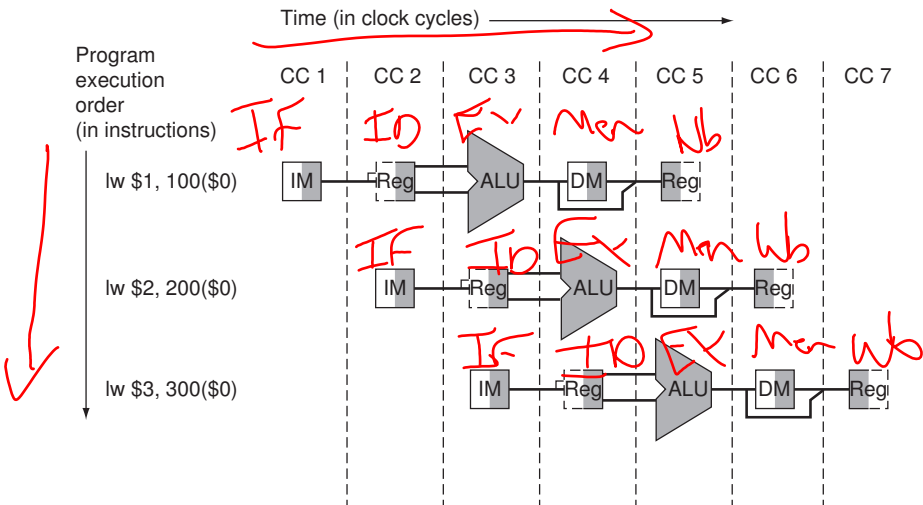
## Pipeline registers

- ▶ The pipeline registers hold the intermediate state of each stage in the pipeline
- ▶ The instructions progress to the next stage when data is latched into the next set of pipeline registers on the next active edge of the clock
- ▶ The *PC* marks the beginning of the Instruction Fetch stage
- ▶ The writing of the register file marks the end of the Write Back stage

## Multi-cycle instruction diagrams

- ▶ It is helpful to visualize the flow of instruction through the pipeline across multiple cycles
- ▶ We can use graphics representing each stage showing where a given instruction is in the pipeline
- ▶ Program order progresses downward, while instructions flow through the pipeline from left to right

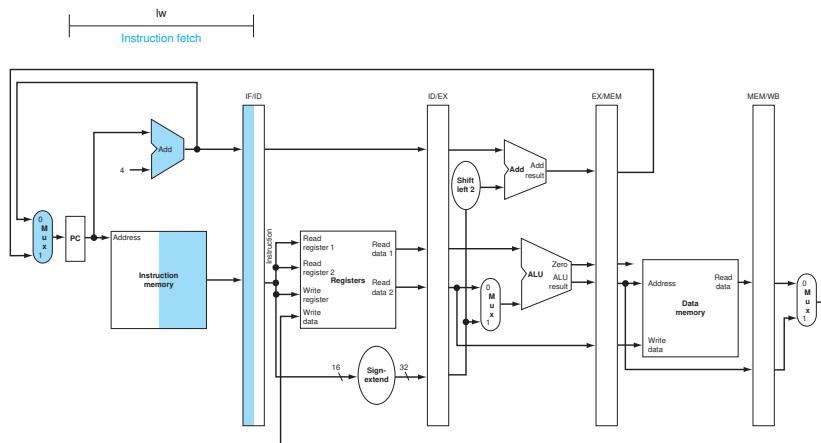
# Multi-cycle instruction diagrams!



## Follow $lw$ through the pipeline

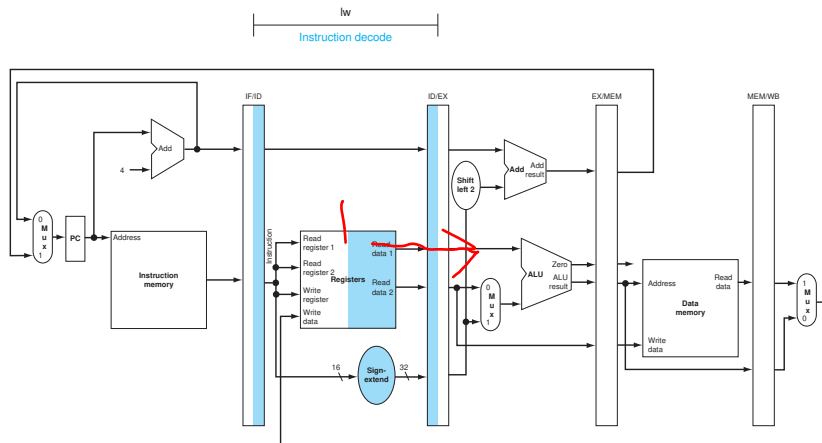
- ▶ Let's following  $lw$  through the pipeline
- ▶ We will look at it's progress 1 stage at a time
- ▶ Using the *single-cycle* timing diagrams with relevant paths highlighted

# /w - Instruction Fetch

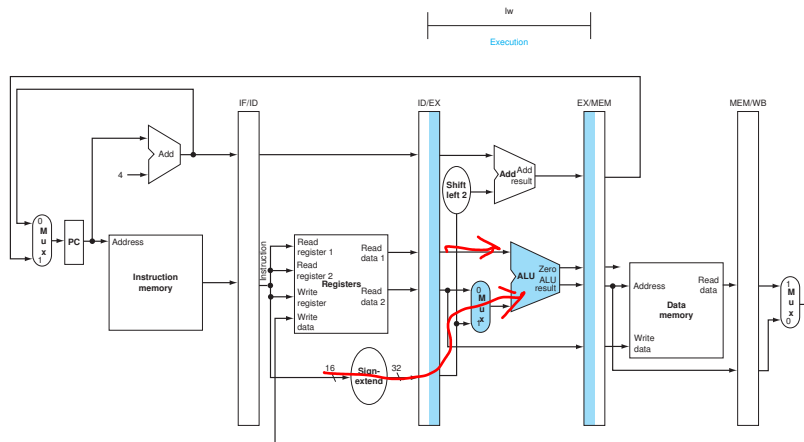




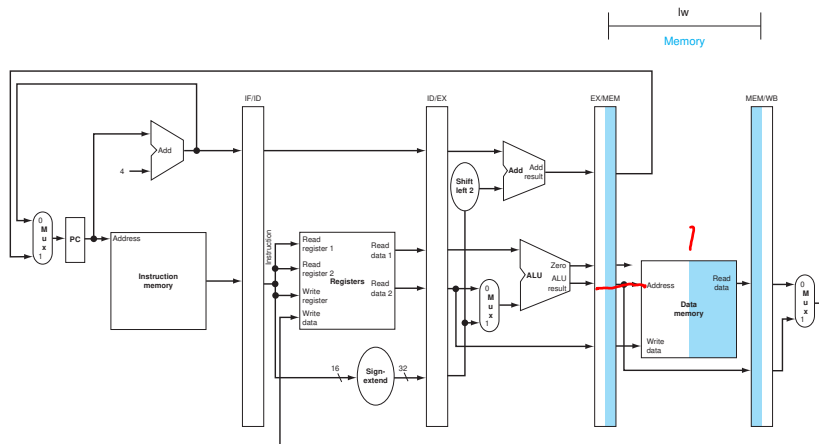
# /w - Instruction Decode



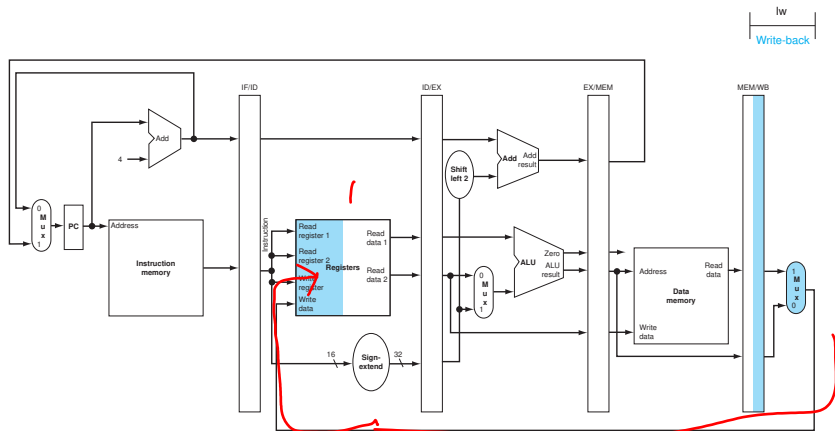
# lw - Execute



# lw - Memory



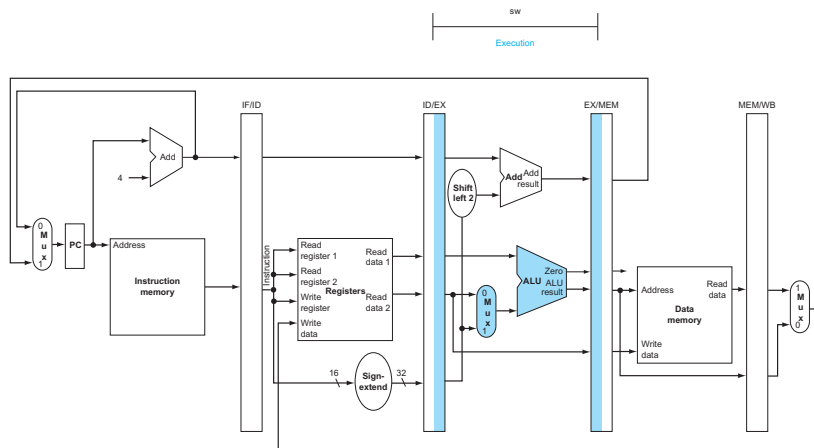
# lw - Write-back



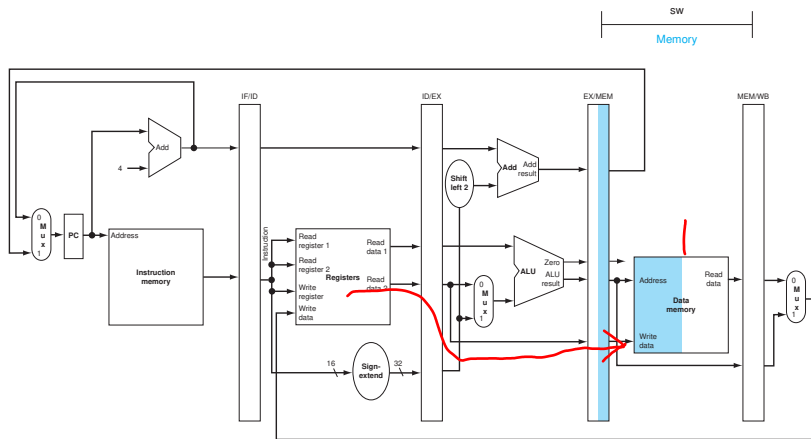
## Why do we look at `lw`??

- ▶ Load word uses the entire pipeline, except for the branch and jump logic
- ▶ For other instructions, such as store word, that is not the case
- ▶ Let's follow `sw` starting in Execute

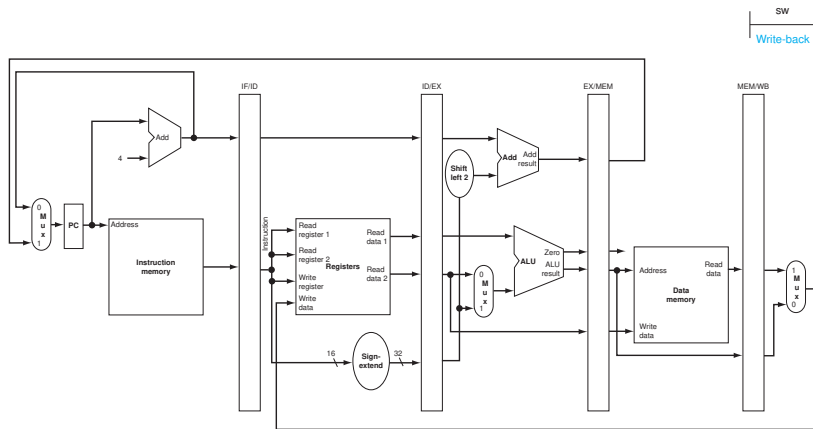
# sw - Execute



# sw - Memory



# sw - Write-back

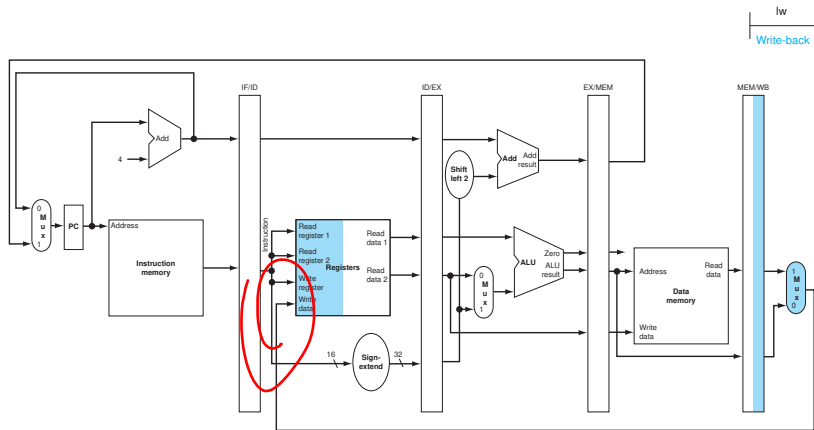




## Notes on *sw*...

- ▶ Notice the *left* half of the memory was highlighted indicating *sw* is writing to memory
- ▶ Nothing happens in the *Write-back* stage!

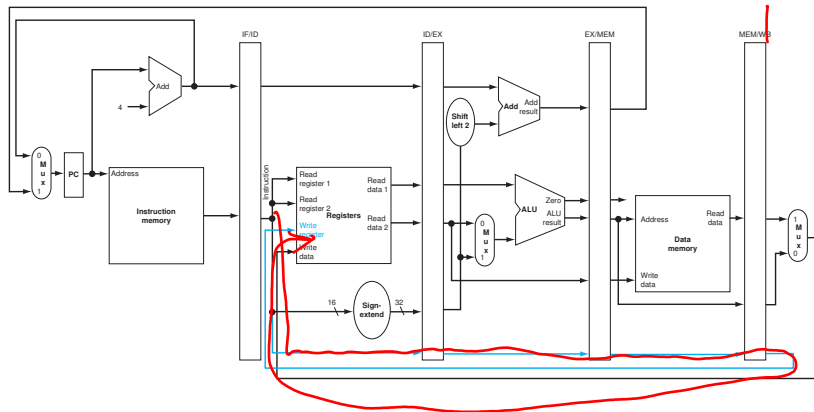
# There is an error in this diagram!



## How to correct the diagram??

- ▶ The **write register** signal is incorrectly routed!
- ▶ It will cause the wrong register to be written
- ▶ The **write register** signal must propagate down the pipeline with the corresponding data

# The corrected diagram...



## Multicycle pipeline diagrams revisited...

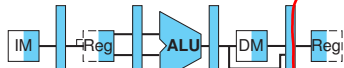
- ▶ Again to visualize the flow of instructions, we like multicycle pipeline diagrams
- ▶ They provide less detail compared to the single-cycle diagram
- ▶ We can look down a column to see what instructions are in the pipeline and where for a given clock cycle

Time (in clock cycles) →  
 CC 1    CC 2    CC 3    CC 4    CC 5    CC 6    CC 7    CC 8    CC 9

Program  
 execution  
 order  
 (in instructions)

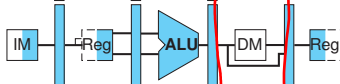
1

lw \$10, 20(\$1)



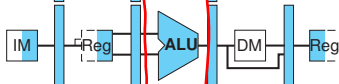
2

sub \$11, \$2, \$3



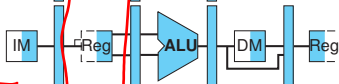
3

add \$12, \$3, \$4



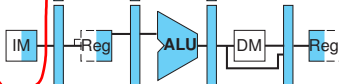
4

lw \$13, 24(\$1)



5

add \$14, \$5, \$6

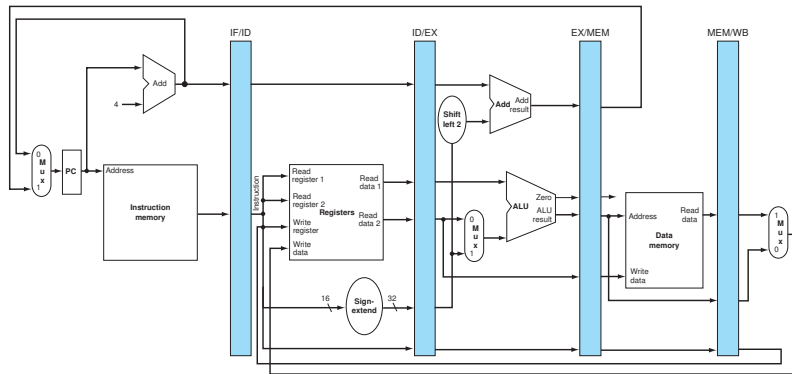


newest →

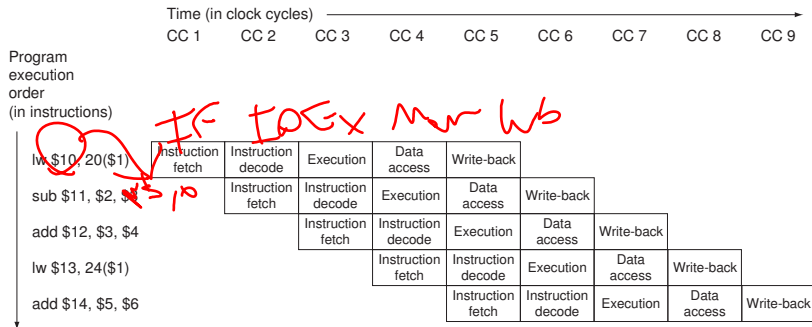
← oldest

# A look at cycle five

add \$14, \$5, \$6	lw \$13, 24(\$1)	add \$12, \$3, \$4	sub \$11, \$2, \$3	lw \$10, 20(\$1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



# A more condensed, easier to draw diagram



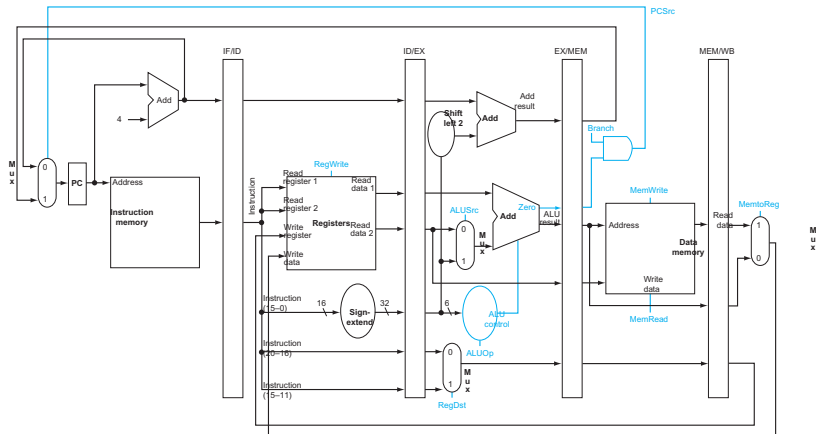


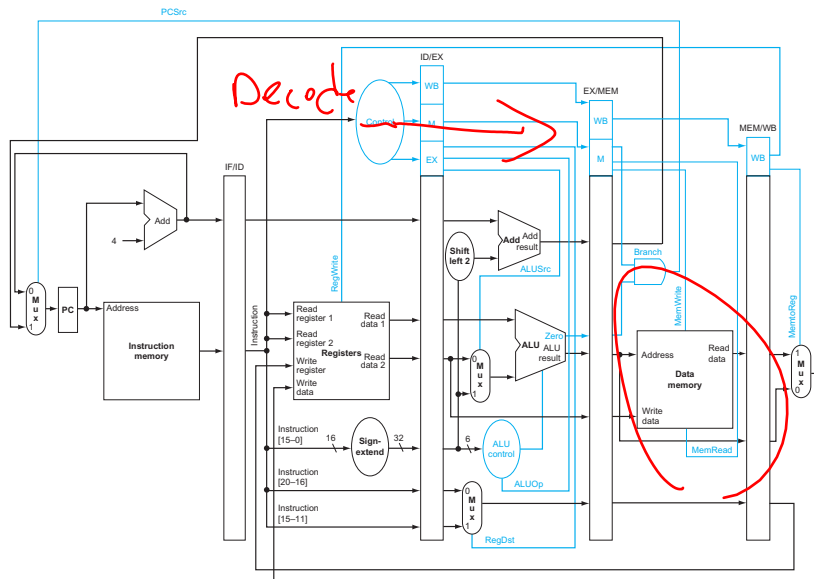
CC	1	2	3	4	5	6
lw	IF	ID	Ex	Mem	WB	
sub		IF	ID	ID	Ex	Mem
add			IF	ID	ID	Ex
lw					IF	ID
add						IF

## Ok so what about control??

- ▶ So far, we have only looked at pipelining the datapath
- ▶ A microarchitecture is made up of datapath and *control*
- ▶ How do we control our 5-stage pipeline?

# Add the muxes and control signals...



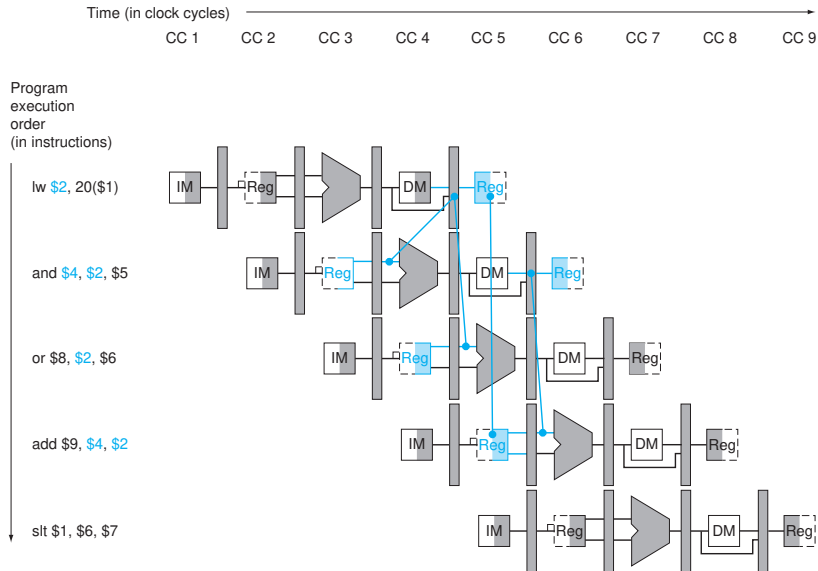


## Control signals...

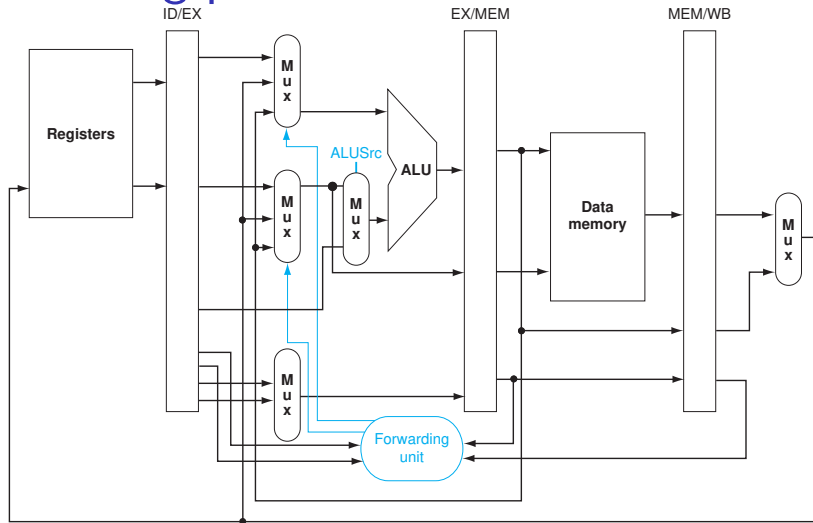
- ▶ Are generated in the exact same way as our single-cycle design
- ▶ Flow down the pipeline with the data of the instruction!
- ▶ Are used in different stages depending on where the datapath component they are controlling is located

## Ok so what about forwarding??

- ▶ For our simple pipeline, data can be forwarded from 1 of 2 locations in the pipeline
- ▶ Data is forwarded to the input of the ALU
- ▶ The register file is written in the beginning of the cycle and read at the end to provide additional bypassing



# Forwarding paths





## Problems with the previous diagram?

- ▶ It's inefficient to have the forwarding detection logic in the *EX* stage. Why?
- ▶ It lengthens the critical path...
- ▶ In lab, we will put our forwarding detection logic in the *Decode* stage

# Assume the following mux selection bits

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

## Forwarding detection logic in *EX*

```
if(ID_EX_Rs != 0 &&  
    EX_MEM_WriteReg == ID_EX_Rs &&  
    EX_MEM_RegWrite) ForwardA = 2'b10;  
else if(ID_EX_Rs != 0 &&  
    MEM_WB_WriteReg == ID_EX_Rs &&  
    MEM_WB_RegWrite) ForwardA = 2'b01;  
else Forward = 2'b00;
```

## Some things to note about the logic...

- ▶ Note the zero check
- ▶ Note that the priority matters!

Consider this example:

```
add $t1, $t1, $t2
```

```
add $t1, $t1, $t3
```

```
add $t1, $t1, $t4
```

- ▶ If no hazard is detected, no forwarding happens
- ▶ The logic for the B side would look similar but would use ID\_EX\_Rt and control ForwardB

## In lab...

- ▶ Detection will happen in *Decode* stage.  
First condition will look similar to this:

```
if(IF_ID_Rs != 0 &&  
    ID_EX_WriteReg == IF_ID_Rs &&  
    EX_RegWrite)
```

- ▶ We will combine input from the **Decode Unit** to control 4-input Muxes in the *Execute* stage

beg loop

SW

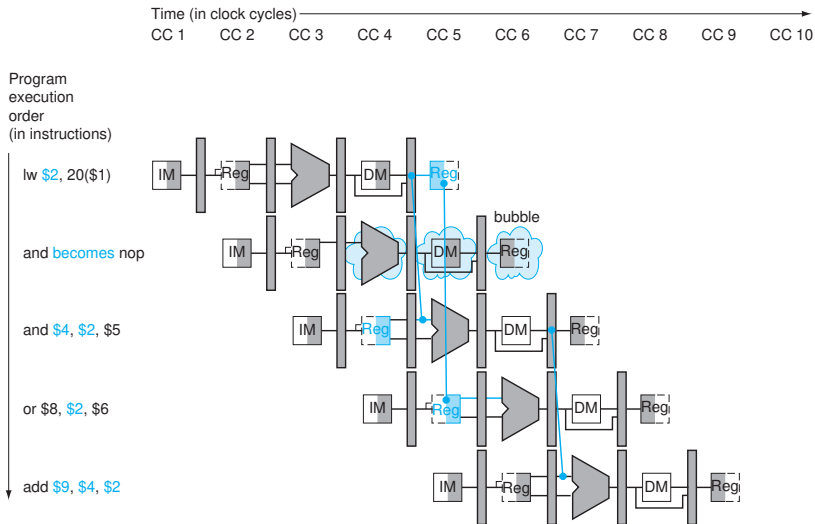
SH

beg	IC	ID	EX
SW		IF	ID
SH			IF

loop: xor

## If you recall...

- ▶ Forwarding is not enough to resolve data hazards
- ▶ We must also build in provisions to *slip* the pipeline
- ▶ Terminology warning: the book uses the term *stall* but to be pedantic we are actually *slipping* the pipeline. This term was used in the early MIPS documentation



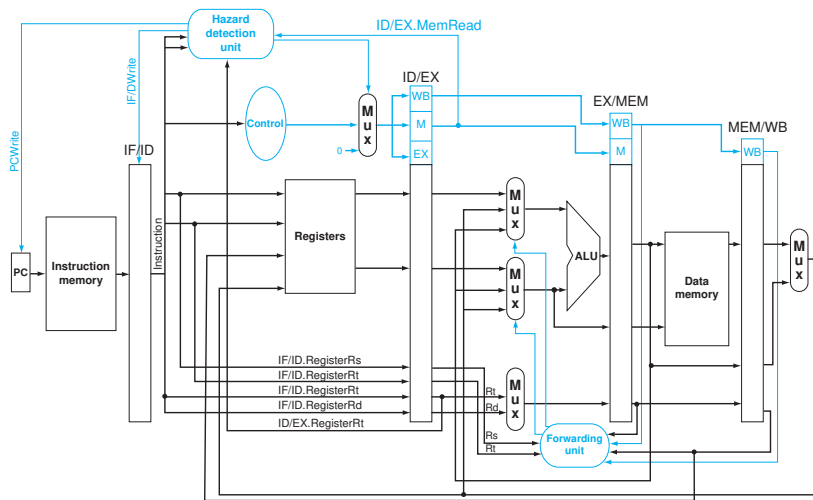


## How to *slip* the pipeline

- ▶ A *slip* is when we stall the front end of the pipeline, while allowing the back end to progress
- ▶ This has the effect of insert nops between dependent instructions
- ▶ To do so, we must add control logic to detect the hazard as well as control signals to the *PC* and *IF/ID* registers

## But wait, there's more!

- ▶ When we hold an instruction in *Decode*, we need to propagate nops
- ▶ We do so by zeroing out control signals that change the state of the processor
- ▶ For example, **MemWrite**, **RegWrite**, *Branch*



## In real life...

- ▶ The previous diagram showed a mux that would propagate 0s for **MemWrite**, **RegWrite**, and *Branch*
- ▶ In actuality, we can accomplish this with a synchronous clear on the appropriate flip-flops in the ID/EX registers

## In real life...

- ▶ The fanout of the clear signal could get large as the state of the processor grows
- ▶ To combat this fanout, we could just propagate a *InstrInvalid* signal through the pipeline
- ▶ Then we gate state change with the logical inverse of this signal



## Logic to detect the */w* hazard

```
if (ID_EX_MemRead && ID_EX_Rt != 0 &&  
    ((ID_EX_Rt == IF_ID_Rs) ||  
     (ID_EX_Rt == IF_ID_Rt))) slip pipeline
```

- ▶ What is missing?
- ▶ *rs* and *rt* are not used by all instructions so our logic above must account for that!
- ▶ Otherwise, we will unnecessarily slip the pipeline

## So far we have...

- ▶ Showed how to handle structural hazards in our pipeline by duplicating logic
- ▶ Showed how to handle data hazards by forwarding and slipping the pipeline



## Now on to *control* hazards!

- ▶ The simplest method is *assume Not Taken*
- ▶ Recall we must *flush* instructions
- ▶ How do we do that in our pipeline?

## How to *flush* properly

- ▶ Just like in the case of inserting nops we need to clear certain control signals
- ▶ Or if we are propagating an invalid signal, we would assert it

## How do we support branch delays?

- ▶ Easy! Simply don't flush the instruction that follows the branch
- ▶ What if a branch is in the branch delay slot??
- ▶ Just don't do it! The MIPS manual specifically says nondeterministic things will happen so just don't do it!

## Which instructions could go in the delay slot?

```
    addi $t2, $t1, 60
loop:
    lw $t4, 0($t1)
    lw $t5, 4($t1)
    xor $t6, $t4, $t5
    sw $t6, 8($t1)
    addi $t1, $t1, 12
    bne $t1, $t2, loop
```

## Scheduling the branch delay slot

```
        addi $t2, $t1, 60
loop:
        lw  $t4, 0($t1)
        lw  $t5, 4($t1)
        xor $t6, $t4, $t5
        addi $t1, $t1, 12
        bne $t1, $t2, loop
        sw  $t6, 8($t1)
```



# Scheduling the branch delay slot

- ▶ The `sw` in the previous example is a good candidate because it will execute regardless of the condition of the branch and does not affect the branch
- ▶ The `addi` is not a good candidate. Why?

## Static branch prediction

- ▶ We could easily modify our simple pipeline to support static prediction techniques
- ▶ *Backwards-Taken, Forward-Not-Taken* simply examines sign of the branch offset field
- ▶ Branch-likely instructions use the opcode to “predict” where to go
- ▶ We must add a recovery mechanism to the pipeline



## Dynamic branch prediction

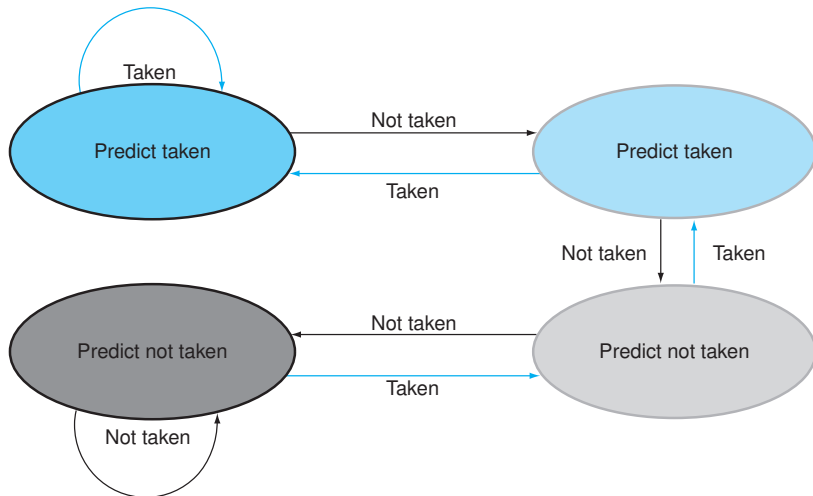
- ▶ In the 90's, lots of research on dynamic branch prediction came out
- ▶ The simplest predictor is a 1-bit wide *branch history table*
  - ▶ Address the table with the lower bits of the word address (ignoring the lower 2-bits of the PC)
  - ▶ Keep track of the last outcome of the branch
  - ▶ If the last outcome was Taken, predict Taken
  - ▶ Branch *aliasing* happens...
- ▶ An improvement is to use a 2-bit wide table



## 2-bit Predictor

- ▶ Requires two mispredictions to change the prediction (hysteresis)
- ▶ Often called a *loop predictor*
- ▶ Simply make the table 2 bits wide and add some state logic

## 2-bit FSM



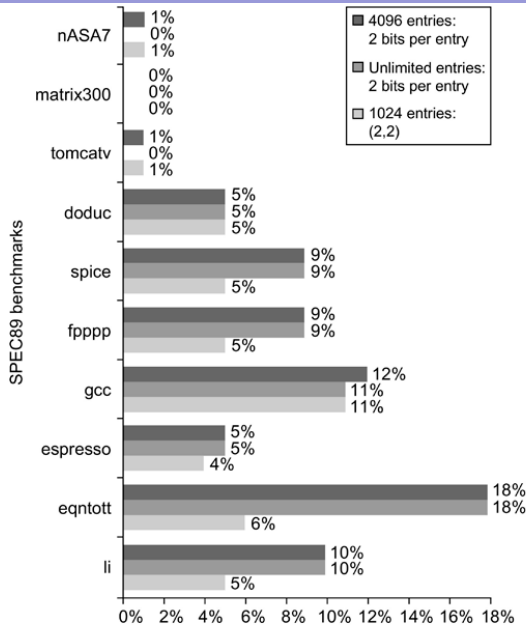
## Accuracy of the 2-bit predictor

- ▶ If our loop iterates 10 times, we will predict correctly 9/10 times (i.e. 90%)
- ▶ This works well for loops but does nothing for if-else like structures
- ▶ *correlated predictors* incorporate *global* history in addition to *local* history

## Correlated predictors??

- ▶ The 2-bit branch history table only predicts based on the *local* history of the current branch
- ▶ A correlated predictor also includes a *branch shift register* that keeps track of the outcome of the last branches executed prior to the current branch, referred to as *global* history
- ▶ This shift register is then concatenated with the bits from the PC to index our history table



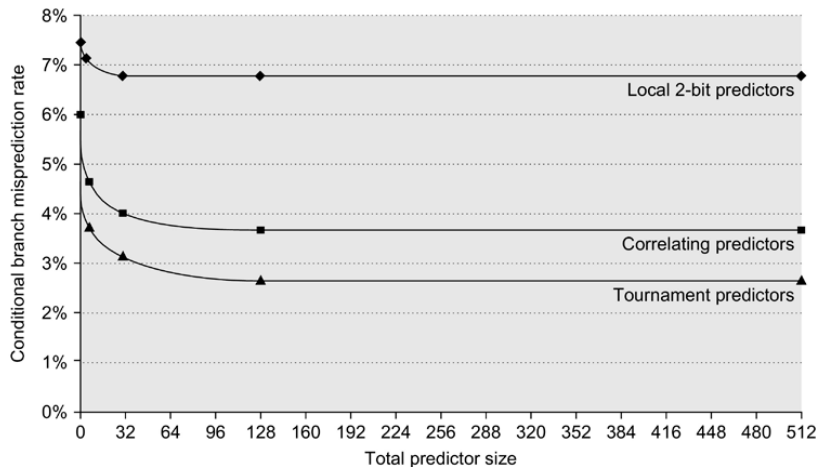




# Can we do better?

- ▶ Yes we can!
  - ▶ some branches do better with just local history (loops) some do better with both global and local history (if-else statements)
  - ▶ We can use a predictor structure to select the correct predictor!
  - ▶ This is called a *tournament predictor*
- ▶ Machine learning techniques
  - ▶ Daniel Jiménez and his perceptron predictor!





# On to Exceptions!

- ▶ An **exception** is an unscheduled interruption on instruction flow
- ▶ Exceptions can be caused by internal or external events
- ▶ An arithmetic overflow during an add is an example of an internal exception
- ▶ An **interrupt** is an example of an external exception

# Internal Exceptions

- ▶ What are some other exceptions?
  - ▶ Undefined instruction
  - ▶ address error on fetch/load
  - ▶ address error on store
  - ▶ bus error
  - ▶ break
  - ▶ syscall
- ▶ There are a few more that have to do with *virtual memory* support, which will be covered in chapter 5

## How does MIPS handle exceptions?

- ▶ Stopping execution of the offending instruction
- ▶ Storing the PC of the offending instruction in the **EPC**
- ▶ Storing a cause code in the **cause** register indicating what exception was thrown
- ▶ Directing execution to an exception address

# Vectored Interrupts

- ▶ Newer versions of MIPS support **vectored interrupts**
- ▶ After an exception is thrown, the processor begins execution at an address based on the cause of the exception
- ▶ Allows the system code to more quickly respond to interrupts

## More about exceptions

- ▶ non-vectored interrupts still have different addresses for reset and TLB misses (chapter 5 material!)
- ▶ Pipelining introduces some challenges when it comes to exceptions
- ▶ Maintaining *precise exceptions* is challenging



# Precise exceptions

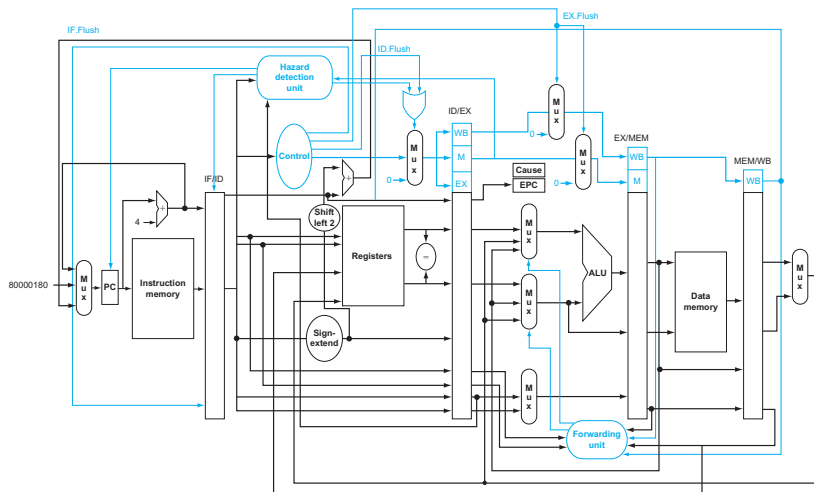
- ▶ Being able to precisely associate an exception with the correct instruction
- ▶ And not allowing that instruction and all instructions after it in program order to commit state
- ▶ Easy to do in a single-cycle design. Why??

# Pipelining and exceptions

- ▶ With pipelining, instructions after the offending instruction have already begun moving through the pipeline when the exception is handled
- ▶ The control logic must ensure those instructions do not commit *state*, i.e. write to register file or memory

# The strategy

- ▶ Detect exceptions at various stages in the pipeline
- ▶ Handle the exceptions right before state must be committed
  - ▶ In the execute stage maybe?
  - ▶ Flush offending instruction and all other instructions after it in the pipeline
  - ▶ Save off EPC and Cause
  - ▶ Redirect the program counter



## Precise exceptions again...

- ▶ Are harder to maintain with floating point and other long latency operations
- ▶ Require the use of a Reorder Buffer for out-of-order execution!
- ▶ Are absolutely necessary when using Virtual memory

# Instruction-Level Parallelism (ILP)

- ▶ Pipelining is technique for exploiting ILP
- ▶ ILP can typically provide a performance boost without changing code
- ▶ More aggressive techniques include *superpipelining*, *superscalar*, and *out-of-order execution*
- ▶ Modern, high-performance processors such as the Intel i7 employ all of the above!

# Superpipelining

- ▶ The MIPS R4000 was an early example
  - ▶ Instruction execution was broken up into 8-stages as opposed to 5-stages
  - ▶ 2 stages for fetch, 2 stages data cache access and a 3<sup>rd</sup> stage for tag compare (Ch. 5)
  - ▶ Achieved a higher clock rate at the expense of a slightly higher CPI
- ▶ Modern processors are 14 stages or deeper...

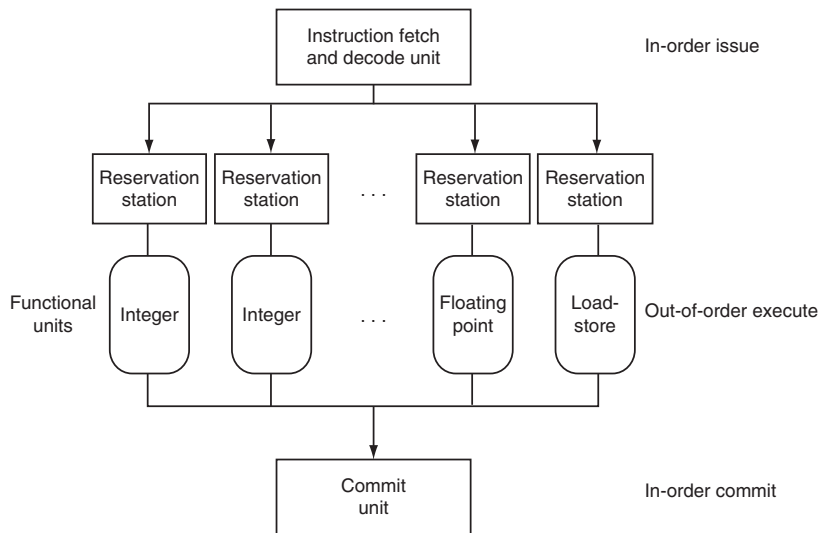
## *Superscalar*

- ▶ Refers to fetching, executing, and committing multiple instructions per cycle
- ▶ Allows for a CPI of less than 1 (i.e. IPC greater than 1)
- ▶ The Intel Pentium I (1993) was an example
  - ▶ Fetched 2 instructions per cycle
  - ▶ 5 stage CISC pipeline
  - ▶ 1 floating point pipeline, 2 integer pipeline
- ▶ Modern processors fetch up to 4 instructions per cycle



## *out-of-order execution*

- ▶ Allow instructions to run out of program order
- ▶ Hides latency of longer instructions or memory access
- ▶ Facilitates the use of more *functional units* for aggressive ILP



Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128–1024 KiB	256 KiB
3rd level cache (shared)	–	2–8 MiB